Strong Replica Consistency for Fault-Tolerant CORBA Applications

P. Narasimhan, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106. USA.
priya@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

The Eternal system provides transparent fault tolerance for CORBA applications, without requiring modifications to the application or the ORB, and without requiring special skills of the CORBA application programmers. Eternal maintains strong replica consistency as replicas of objects perform operations, and even as they fail and are recovered. Eternal implements the new Fault Tolerant CORBA standard.

1 Introduction

The Common Object Request Broker Architecture (CORBA) [12] middleware supports applications that consist of objects distributed across a system, with client objects invoking server objects, which return responses to the client objects after performing the requested operations. CORBA's Object Request Broker (ORB) acts as an intermediary in the communication between a client object and a server object, shielding them from any differences in their programming languages and their physical locations. CORBA's TCP/IP-based Internet Inter-ORB Protocol (IIOP) sustains this communication even if the client object and the server object use different operating systems, byte orders and hardware architectures.

The Eternal system [9] enhances CORBA by providing fault tolerance for CORBA applications, without requiring the application programmer to be concerned with the difficult issues of fault tolerance. The value of Eternal in developing fault-tolerant CORBA applications lies in the *transparency* of its approach, *i.e.*, neither the application code nor the CORBA middleware needs to be modified to benefit from Eternal's fault tolerance. The transparency of Eternal allows existing CORBA applications to be rendered fault-tolerant easily and quickly, by application programmers who have no special experience or training in fault tolerance.

Another key feature of Eternal is that it provides *strong replica consistency*, even as objects and the processors that host them fail, even as object receive and process invocations, and even as objects are created and destroyed.

2 Strong Replica Consistency

Eternal provides fault tolerance for CORBA applications by replicating the objects of the application. The purpose of replication is to provide multiple, redundant, identical copies, or *replicas*, of an object so that the object can continue to provide useful services, even though some of its replicas fail, or as the processors hosting some of its replicas fail. The essence of replication is that the replicas of an object must be consistent in state.

For ensuring strong replica consistency, Eternal requires application objects to be *deterministic* (or to be rendered deterministic) in their behavior so that if two replicas of an object start from the same initial state, and have the same sequence of messages applied to them, in the same order, then the two replicas will reach the same final state. In cases where the application is inherently non-deterministic, *e.g.*, the application uses system-specific or processor-specific functions, Eternal provides mechanisms to identify and "sanitize" such sources of non-determinism, thereby rendering the application deterministic from the viewpoint of replication. The different components of Eternal interact to address the challenges in maintaining strong replica consistency.

- **Ordering of operations.** All of the replicas of each replicated object must perform the same sequence of operations in the same order to achieve replica consistency. Eternal achieves this by exploiting a reliable totally-ordered multicast protocol for conveying the IIOP invocations (responses) to the replicas of a CORBA server (client).
- **Duplicate operations.** Replication, by its very nature, can lead to duplicate operations. For example, if every replica of a three-way replicated client object invokes a method of a replicated server object, every server replica will receive three copies of the same invocation, one from each of the client replicas. Eternal ensures that such duplicate invocations (responses) due to a replicated client (server) object are filtered so that the server (client) object receives only a single copy of every distinct invocation (response).
- **Recovery.** When a new replica is activated, or when a failed replica is recovered, *before* it can start to operate, it must have the same state as the other replicas of the object. Eternal retrieves the state from an existing operational replica of the object, and transfers the state to the new or recovering replica before making it operational.
- **Multithreading.** Replicas of a multithreaded CORBA object may become inconsistent if the threads, and the operations that they execute, are not carefully controlled. For multithreaded ORBs or objects that allow the simultaneous execution of multiple operations, Eternal provides mechanisms to ensure strong replica consistency, regardless of the multithreading of the ORB or the application.

3 Architecture of the Eternal System

Eternal's OMG-compliant fault tolerance infrastructure consists of components *above* the ORB and mechanisms *below* the ORB,

0-7695-1068-X/01 $10.00 © 2001 IEEE

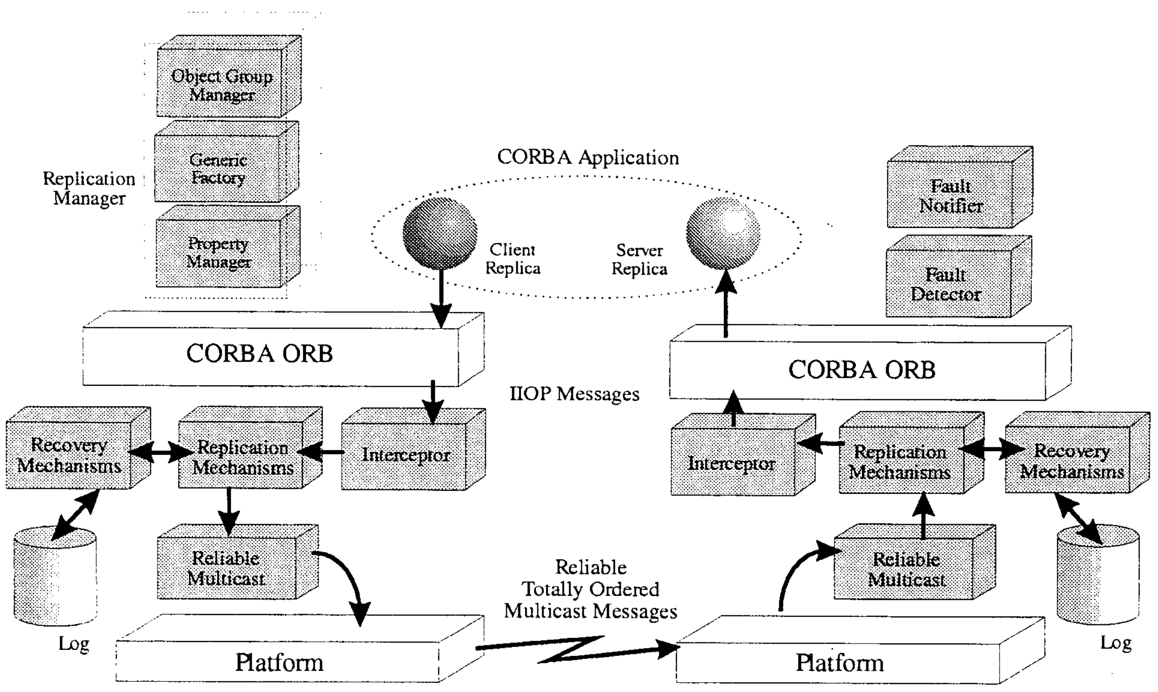


Figure 1: Eternal's OMG-compliant fault tolerance infrastructure, consisting of components above and below the ORB.

as shown in Figure 1. Because Eternal's components above the ORB are composed of CORBA objects, they can also be replicated (just as the application's CORBA objects are) by Eternal, with an adequate number of replicas distributed across the processors in the system. On the other hand, Eternal's Mechanisms, underneath the ORB, are composed of non-CORBA C++ objects, and must be present on every processor that is to host replicated objects.

Using Eternal's fault tolerance infrastructure, both client and server objects of the CORBA application can be replicated, with the replicas distributed across the system. Eternal supports different replication styles – active, cold passive and warm passive replication – for every application object. To facilitate replica consistency, the Eternal infrastructure conveys the IIOP messages of the CORBA application using the reliable totally-ordered multicast messages of the underlying Totem system [8].

The Replication Manager component replicates each application object, according to user-specified requirements, and distributes the replicas across the system. The Fault Detector component detects the failure of replicas, objects and processors in the system. The Fault Notifier component uses the information gathered by the Fault Detector to notify other interested components of faults that have occurred in the system. The Replication Manager, the Fault Detector and the Fault Notifier are themselves implemented as collections of CORBA objects and, thus, can benefit from Eternal's fault tolerance.

The Interceptor captures the IIOP messages (containing the client's requests and the server's replies), which are intended for TCP/IP, and diverts them instead to the Replication Mechanisms for multicasting via Totem. The Replication Mechanisms, together with the Recovery Mechanisms, maintain strong consistency of the replicas, detect and recover from faults, and sustain operation in all components of a partitioned system, should a partition occur.

The types of faults tolerated by Eternal, and the underlying Totem system, are communication faults, including message loss and network partitioning, and processor, process and object faults. Eternal can also tolerate arbitrary faults [10] by exploiting protocols with more stringent guarantees than Totem provides.

3.1 Replication Management

To manage the replication of an object, Eternal employs the notion of an object group, where the members of the group correspond to the replicas of an object. In Eternal, both client and server objects can be replicated and, thus, constitute object groups.

The Replication Manager is a crucial component of the Eternal's fault tolerance infrastructure, and handles the creation, deletion and replication of both the application objects and the infrastructure objects. The Replication Manager component replicates objects, and distributes the replicas across the system. Although each replica of an object has an individual object reference, the Replication Manager component fabricates an object group reference for the replicated object that clients use to contact the replicated object. The Replication Manager's functionality is achieved through the Property Manager, Generic Factory and Object Group Manager components.

The Property Manager component allows a user to assign values to a number of fault tolerance properties for every application object that is to be replicated. Eternal provides the user with the flexibility to configure the replication of every application object by assigning the values of various fault tolerance properties, including:

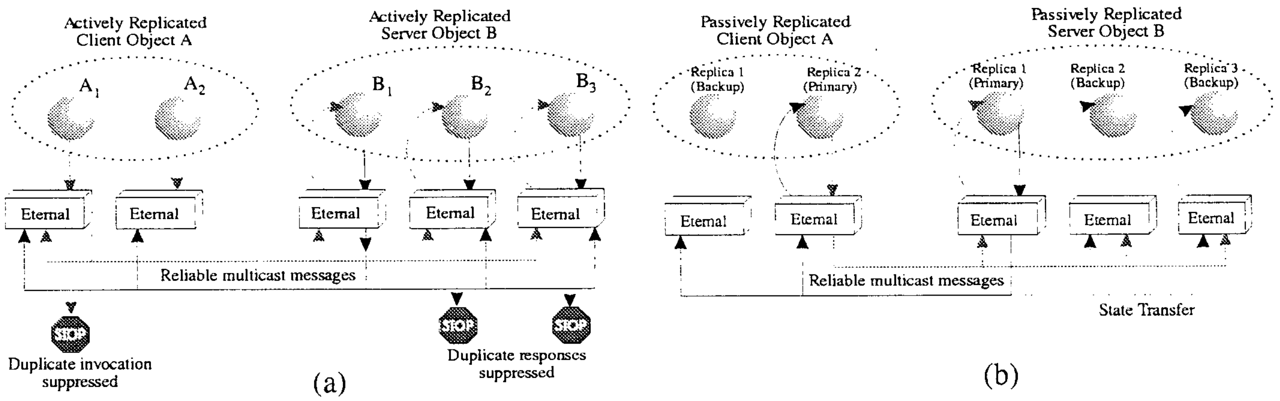

Actively Replicated Client Object A
Actively Replicated Server Object B
A1
A2
B1
B2
B3
Eternal
Reliable multicast messages
Duplicate invocation suppressed
Duplicate responses suppressed
(a)
Passively Replicated Client Object A
Passively Replicated Server Object B
Replica 1 (Backup)
Replica 2 (Primary)
Replica 1 (Primary)
Replica 2 (Backup)
Replica 3 (Backup)
Reliable multicast messages
State Transfer
(b)

Figure 2: Two of the Replication Styles supported by the Eternal system (a) active replication and (b) warm passive replication.

- **Replication Style** – stateless, actively replicated, cold passively replicated or warm passively replicated. Eternal's support for active replication and warm passive replication are shown in Figure 2.
- **Membership Style** – addition, or removal, of an object's replicas is application-controlled or infrastructure-controlled.[1]
- **Consistency Style** – replica consistency (including recovery, checkpointing, logging, etc.) is application-controlled or infrastructure-controlled.
- **Factories** – objects that create and delete the replicas of the object.
- **Initial Number of Replicas** – the number of replicas of the object to be created initially.
- **Minimum Number of Replicas** – the number of replicas of the object that must exist for the object to be sufficiently protected against faults.
- **Checkpoint Interval** – the frequency at which the state of an object is to be retrieved and logged for the purposes of recovery.

The Generic Factory component allows users to create replicated objects in the same way that they would normally create unreplicated objects. The Generic Factory interface is inherited by the Replication Manager component to allow the application to invoke the Replication Manager directly to create and delete replicated objects. When asked to create a replicated object through its Generic Factory interface, the Replication Manager component, in turn, delegates the operation to the factories on the processors where the individual replicas of the object are to be created.

The Object Group Manager component allows users to control directly the creation, deletion and location of individual replicas of an application object. While this violates replication transparency (because the user is explicitly aware of the replicas of an object), and must be used with care to ensure replica consistency, it is useful for expert users who wish to exercise direct control over the replication of application objects.

Infrastructure-controlled Membership Style, in conjunction with infrastructure-controlled Consistency Style, is favored for the development of fault-tolerant CORBA applications, because it provides the maximal ease of use and transparency to the application, with the assurance of strong replica consistency, which the Eternal infrastructure maintains under both fault-free and recovery conditions.

[1]"Infrastructure-controlled" is equivalent to "Eternal-controlled".

3.2 Fault Detection and Notification

The Fault Detector component is capable of detecting host, process and object faults. Each application object inherits a `Monitorable` interface to allow the Fault Detector component to determine the object's status. The Fault Detector component communicates the occurrence of faults to the Fault Notifier.

On receiving reports of faults from the Fault Detector component, the Fault Notifier component filters them to eliminate any inappropriate or duplicate reports. The Fault Notifier component then distributes fault event notifications to all of the objects that have subscribed to receive such notifications. The Replication Manager component is one such subscriber.

Eternal allows the user to influence fault detection for an object through the following fault tolerance properties:

- **Fault Monitoring Style** – the object is monitored by periodic "pinging" (pull monitoring) of the object or, alternatively, by periodic "i-am-alive" messages (push monitoring) sent by the object.
- **Fault Monitoring Granularity** – the replicated object is monitored on the basis of an individual replica, a location, or a location-and-type.
- **Fault Monitoring Interval** – the frequency at which an object is to be "pinged" to detect if it is alive or has failed.

As shown in Figure 3, the Fault Detection framework can be structured in a hierarchical way, with the global replicated Fault Detector component triggering the operation of local fault detector components located on each processor. Any faults detected by the local fault detectors are reported to the global replicated Fault Notifier component. The Replication Manager, being a subscriber of the Fault Notifier component, receives reports of any faults that occur in the system, and can initiate appropriate actions to enable the system to recover from the faults.

3.3 Interception

The Eternal Interceptor is a non-ORB-level, non-application-level component that transparently "attaches" itself to every executing CORBA object, without the object's knowledge or the

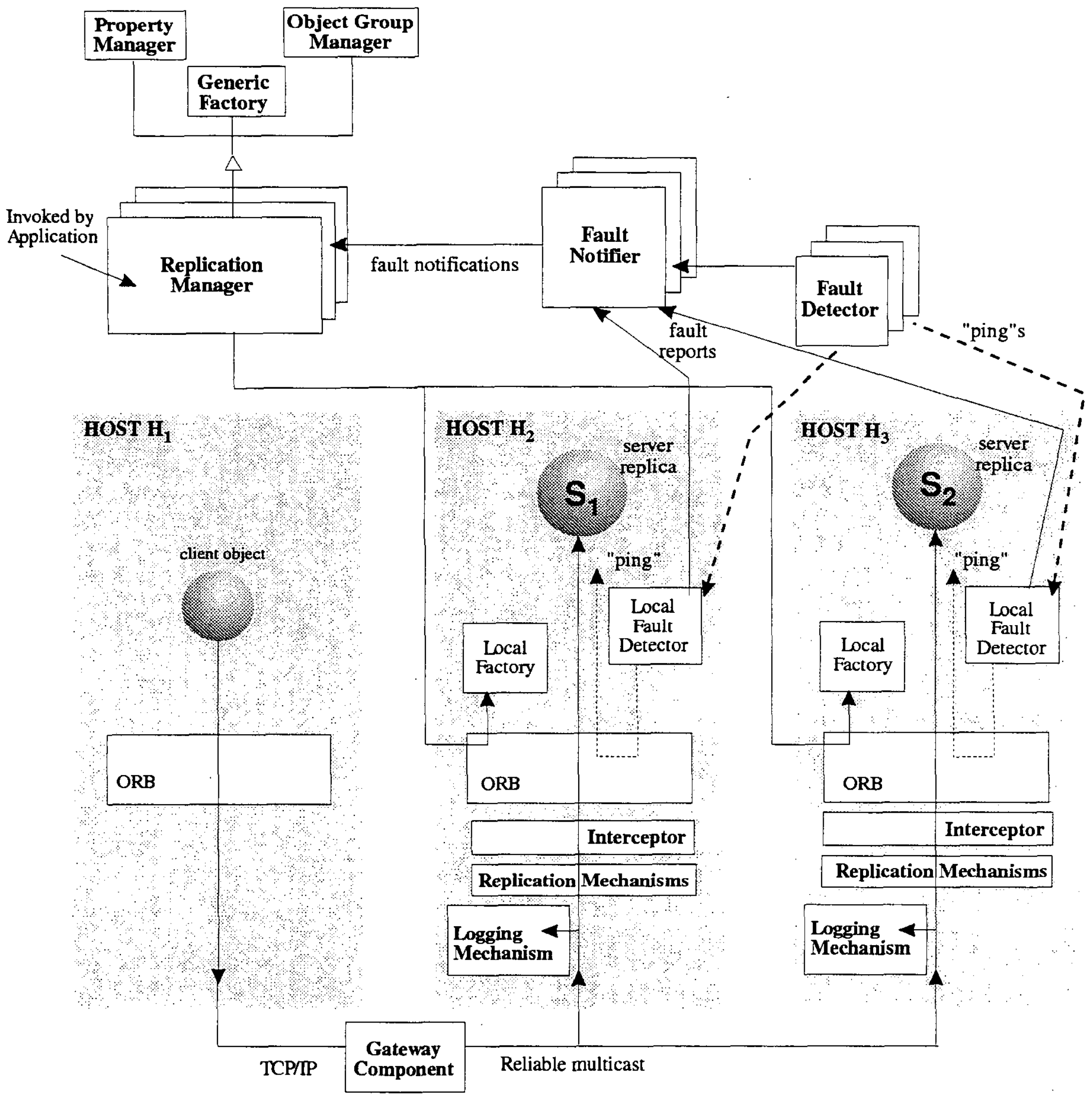


Figure 3: Interaction between Eternal's components below the ORB and above the ORB.

ORB's knowledge, and that can modify the object's behavior as desired. The advantage of Eternal's Interceptor, located underneath the ORB, is not only its transparency to the ORB and to the application, but also its implementation in an ORB-independent manner. This ORB independence allows Eternal to be used with many different ORBs without modifying, or even having access to, the ORB's code.

Current operating systems provide hooks that can be exploited to develop components such as interceptors. With the Unix operating system, at least two possible implementations of interceptors exist. The first of these approaches, the */proc*-based implementation, provides for interception at the level of system calls. The second approach, the library interpositioning implementation (also possible on Windows NT), provides for interception at the level of library routines. While the techniques differ, the intent and use of the interceptor in both cases is identical, and requires no modification of the intercepted CORBA objects, the ORB or the operating system.

The specific system calls to redefine in a */proc*-based implementation or the specific library routines to redefine in a library-interpositioning implementation, depend on the extent of the information that the interceptor must extract (from the ORB or the CORBA application) to enhance the application with new features. The interceptor may capture all, or a particular subset, of the system calls or library routines used by the CORBA application, depending on the feature being added.

Eternal's Interceptor currently employs the library interpositioning approach, because of its lower overheads and ease of deployment with various ORBs. The system calls, or library routines, of interest to Eternal are those associated with the communication of CORBA's IIOP messages. Because the ORB conveys the IIOP messages over TCP/IP, Eternal's Interceptor captures, and redefines, the routines related to TCP/IP communication. The redefinition of these routines allows the Interceptor to divert the captured IIOP messages to the Eternal Replication Mechanisms.

3.4 Replication Mechanisms

The interaction between Eternal's components above the ORB and Eternal's Mechanisms below the ORB, is captured in Figure 3. To facilitate replica consistency, Eternal's Replication Mechanisms (underneath the ORB) convey the IIOP messages of the CORBA application using the reliable totally-ordered multicast messages of the underlying multicast group communication protocol [8].

Eternal's Replication Mechanisms perform different operations for the different replication styles, as shown in Figure 2. For an actively replicated server (client) object, each replica responds to (invokes) every operation. Thus, the Replication Mechanisms deliver every IIOP invocation (response) intended for a replicated server (client) to every server (client) replica through the Interceptor. For active replication, the failure of a single active replica is masked due to the presence of the other active replicas that are also performing the operation.

For a passively replicated server (client) object, only one of the replicas, designated the *primary*, responds to (invokes) every operation. The remaining replicas of the object are referred to as the *backup* replicas. The Replication Mechanisms deliver every IIOP invocation (response) only to the primary replica of a passively replicated server (client) object. In the case of warm passive replication, the backup replicas are synchronized periodically with the primary replica. In the case of cold passive replication, the backup replicas are not loaded, but Eternal periodically retrieves, and stores in a log, the state of the primary replica. In the event that the primary replica fails, one of the backup replicas takes over as the new primary replica.

3.5 Logging and Recovery

Every replicated CORBA object can be regarded as having three kinds of state: *application-level state* (known to, and programmed into the object by, the application programmer) *ORB-level state* (maintained by the ORB for the object) and *infrastructure-level state* (invisible to the application programmer and maintained for the object by Eternal). Application-level state is typically represented by the values of the data structures of the replicated object. ORB-level state is vendor-dependent and consists of the values of the data structures (last-seen request identifier, threading policy, *etc.*). Infrastructure-level state is independent of, and invisible to, the replicated object as well as to the ORB, and involves information that Eternal maintains for consistent replication.

Eternal's Logging-Recovery Mechanisms ensure that all of the replicas of an object are consistent in application-level, ORB-level and infrastructure-level state. State transfer to a new or recovering replica includes the transfer of application-level state to the new replica, ORB-level state to the ORB hosting the new replica, and infrastructure-level state to the Logging-Recovery Mechanisms that manage the new replica.

To enable application-level state to be captured, every replicated CORBA object must inherit from an interface that contains methods for retrieving and assigning an object's state. The interface accesses the object's state in its entirety. The Logging-Recovery Mechanisms invoke this interface to retrieve, or *checkpoint*, the application object's state. In the case of warm passive replication, the Logging-Recovery Mechanisms checkpoint the primary replica's state periodically, and transfer the checkpointed state to the backup replicas. In the case of cold passive replication, the Logging-Recovery Mechanisms store the checkpointed state of the primary replica into a log for restoring the state of a new primarys should the existing primary fail. The frequency of checkpointing is determined on a per-group basis, by the user at deployment time, when the other fault tolerance properties (number of replicas, location of replicas, etc) are also assigned their values.

Because the state retrieval from an existing (primary) active (passive) replica occurs at a different point in the message sequence from the assignment of the retrieved state to the new (backup) active (passive) replicas, the Logging-Recovery Mechanisms at the state retrieval and assignment locations must synchronize the retrieval and state assignment messages. Furthermore, the Logging-Recovery Mechanisms must enqueue all new invocations and responses that arrive for a replica while its state is being assigned.

The Logging-Recovery Mechanisms on a processor are responsible for storing the invocations, responses and state checkpoints of the replicas hosted on that processor. Typically, different object groups exist on a processor and, thus, the Logging-Recovery Mechanisms maintain a single physical log per processor, with the log being indexed by the object group identifier.

Each entry in the log is a log record that contains a received IIOP message along with a special Eternal-specific header associated with the IIOP message for duplicate detection, garbage collection of the log, etc. The records are stored in the log as they arrive at the Logging-Recovery Mechanisms, through the totally ordered message sequence provided by the underlying multicast protocol.

Because the Logging-Recovery Mechanisms have access to the log, they are in a position to match up responses with their corresponding invocations and to detect and suppress duplicate invocations, responses and state transfer messages. To enable incoming response messages to be matched with their corresponding invocations, the Logging-Recovery Mechanisms insert an invocation (response) identifier into the Eternal-specific header for each outgoing IIOP invocation (response) message. For an outgoing response, the Logging-Recovery Mechanisms "remember" and reuse a portion of the invocation identifier associated with the invocation that resulted in this response. The portion of the invocation identifier that is reused in its counterpart response identifier is the *operation identifier*, which represents the operation consisting of the invocation-response pair.

By exploiting the unique totally-ordered sequence numbers that Totem assigns to each message that it delivers, the Logging-Recovery Mechanisms on *different* processors ensure that they always assign the same unique operation identifier for each distinct operation. Also, by performing the duplicate detection and suppression to every incoming message that they receive, the Logging-Recovery Mechanisms ensure that the target application objects receive only one copy of every distinct invocation or response intended for them. Furthermore, the Logging-Recovery Mechanisms perform the duplicate detection *before* recording messages in the log; thus, the log is a sequence of non-duplicate log records.

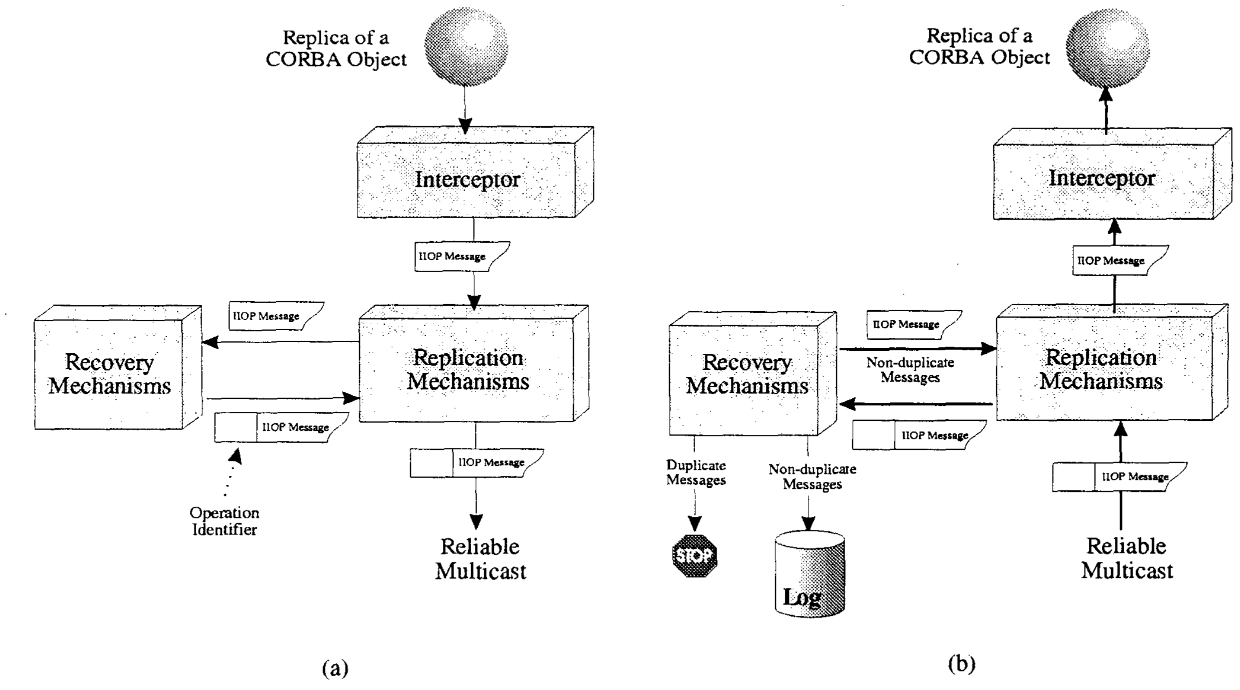


Figure 4: Interaction between Eternal's mechanisms below the ORB for (a) outgoing messages and (b) incoming messages.

4 Consistency under Multithreading

Many commercial ORBs are multithreaded, and multithreading can yield substantial performance advantages. Unfortunately, the specification of multithreading in the CORBA standard does not place any guarantee on the order of operations dispatched by a multithreaded ORB. In particular, the specification of the Portable Object Adapter (POA), which is a key component of the CORBA standard, provides no guarantees on the POA's dispatching of requests to threads. The ORB/POA may dispatch several requests for the same object within multiple threads at the same time.

In addition to ORB-level threads, the CORBA application may itself be multithreaded, with the thread scheduling having been determined by the application programmer. The application programmer must ensure correct sequencing of operations and must prevent thread hazards. Careful application programming can only ensure thread-safe operations within a single replica of an object; however, it does not guarantee that threads and operations are dispatched in the same order across all of the replicas of the object.

Thus, to preserve replica consistency for multithreaded objects, the Eternal system enforces deterministic behavior across all of the replicas of a multithreaded object by controlling the dispatching of threads and operations identically within every replica through a deterministic operation scheduler.

The scheduler dictates the creation, activation, deactivation and destruction of threads, within every replica of a multithreaded object, as required for the execution of the current operation "holding" the logical thread-of-control. Exploiting the thread library interpositioning mechanisms of Eternal's Interceptor, the scheduler can transparently override any thread or operation scheduling performed by either the nondeterministic multithreaded ORB within the replica, or by the replica itself.

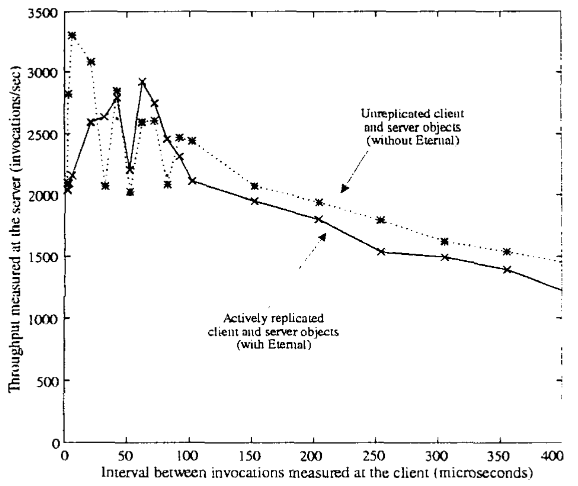
Based on this incoming totally ordered sequence of messages, the scheduler at each replica decides on the immediate delivery, or the delayed delivery, of the messages to that replica. At each replica, the scheduler's decisions are identical and, thus, operations and threads are dispatched identically at each replica, ensuring determinstic operation across all replicas of the object.

4.1 Implementation and Performance

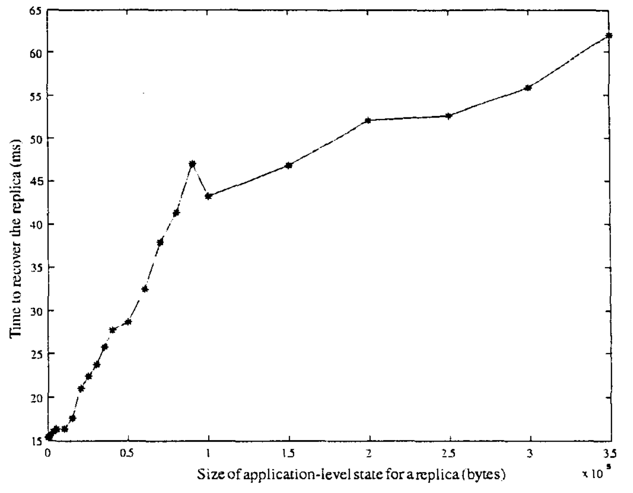
The current implementation of Eternal's fault tolerance infrastructure provides transparent fault tolerance to unmodified CORBA applications running over the following unmodified commercial implementations of CORBA over standard operating systems (Solaris 2.x, Red Hat Linux 6.0 and HP-UX 10.20):

- VisiBroker from Inprise Corporation
- Orbix from Iona Technologies
- e*ORB and CORBAplus from Vertel (previously Expersoft)
- ORBacus from Object-Oriented Concepts, Inc.
- TAO from Washington University, St. Louis
- omniORB2 from AT & T Laboratories, U.K.
- ILU from Xerox PARC

The efficient Totem multicast group communication protocol allows Eternal to provide fault tolerance with minimal overhead. Using Eternal, for Solaris 2.x on 167Mhz SPARC workstations connected by a 100Mbps Ethernet, when application objects are actively replicated, test applications typically incur only a 10% increase in round-trip invocation time, compared with their unreplicated unreliable counterparts. For RedHatLinux 6.0 on



(a)



(b)

Figure 5: (a) Throughputs obtained for an unreplicated application (without Eternal) and its three-way actively replicated counterpart (with Eternal), and (b) Variation of the recovery time for a server replica with the size of the replica's application-level state.

400Mhz Intel Pentium processors connected by a 100Mbps Ethernet, this overhead reduces to 3%.

To measure the performance of Eternal, we used a simple test application developed with the VisiBroker 3.2 ORB. The measurements were taken over a network of six dual-processor 167 MHz UltraSPARC workstations, running the Solaris 2.5.1 operating system and connected by a 100 Mbps Ethernet. The graph in Figure 5(a) shows the throughput obtained for the three-way active replication of both the client and the server objects using Eternal, as compared with the throughput obtained for the unreplicated client and server objects without Eternal.

The performance of the Eternal system during the recovery of a new or failed replica of an object is shown in Figure 5(b). The graph shows the time to recover a server replica in a test application developed with Inprise's VisiBroker 4.0 C++ ORB. The measurements were taken over a network of dual-processor 167 MHz UltraSPARC workstations, running Solaris 2.7, and connected by a 100 Mbps Ethernet. The time to recover such a failed replica was measured as the time interval between the re-launch of the failed replica and the replica's reinstatement to normal operation. The graph shows the recovery times obtained with this test application for varying sizes (from 10 bytes to 350,000 bytes) of the application-level state that is transferred across the network to recover a failed server replica.

5 Related Work

Other systems have been developed that address issues related to consistent object replication and fault tolerance in the context of CORBA. The Electra toolkit [5] built by Maffeis on top of Horus [14] provides support for replicated CORBA objects, as does Orbix+Isis [4] on top of Isis [1]. Both Electra and Orbix+Isis integrate the replication and group communication mechanisms into the ORB, thereby requiring modification of the ORB.

The Object Group Service (OGS) [3] provides replication for CORBA applications through a set of CORBA services. Replica consistency is ensured through group communication based on a consensus algorithm implemented through CORBA service objects. OGS provides interfaces for detecting the liveness of objects, and mechanisms for duplicate detection and suppression, and for the transfer of application-level state.

Newtop is a group communication toolkit that is exploited to provide fault tolerance to CORBA using the service approach. While the fundamental ideas are similar to OGS, the Newtop-based object group service [7] has some key differences. Of particular interest is the way this service handles failures due to partitioning – support is provided for a group of replicas to be partitioned into multiple sub-groups, with each sub-group being connected within itself. No mechanisms are provided, however, to ensure consistent remerging of the sub-groups once communication is reestablished.

The Maestro toolkit [15] includes an IIOP-conformant ORB with an open architecture that supports multiple execution styles and request processing policies. The replicated updates execution style can be used to add reliability and high availability properties to client/server CORBA applications in settings where it is not feasible to make modifications at the client side, as is the case for unreplicated clients wishing to contact replicated objects.

The AQuA architecture [2] is a dependability framework that provides object replication and fault tolerance for CORBA applications. AQuA exploits the group communication facilities and the ordering guarantees of the underlying Ensemble and Maestro toolkits to ensure the consistency of the replicated CORBA objects. AQuA supports both active and passive replication, with state transfers to synchronize the states of the backup replicas with the state of the primary replica in the case of passive replication.

The Distributed Object-Oriented Reliable Service (DOORS) [11] provides fault tolerance through a service approach, with CORBA objects that detect, and recover from, replica and processor faults. The system provides support for resource management based on the needs of the CORBA application. DOORS employs libraries for the transparent checkpointing of applications; however, duplicate detection and suppression are not addressed.

The Interoperable Replication Logic (IRL) [6] also provides fault tolerance for CORBA applications through a service approach. One of the aims of IRL is to uphold CORBA's interoperability by supporting a fault-tolerant CORBA application that is composed of objects running over implementations of CORBA from different ORB vendors.

6 Conclusion

The Eternal system provides transparent OMG-compliant fault tolerance for CORBA applications, requiring no modifications to either the application or the CORBA middleware. Eternal's components and mechanisms interact to provide strong replica consistency for every replicated object of the CORBA application. Recognizing that real-world applications are necessarily multi-tiered (i.e., containing objects that play the roles of both client and server), Eternal supports the replication of both client and server objects.

Eternal's transparency reduces the time to develop a new fault-tolerant application, requires no retraining of application programmers, and enables existing applications to be made fault-tolerant. With shorter time-to-market and higher reliability being critical to many different kinds of applications, these applications can be provided the fault tolerance that they need more quickly, more affordably and more reliably using Eternal.

Our experience in designing and building the Eternal system led to our active participation in developing the new standard for Fault-Tolerant CORBA [13] that the Object Management Group (OMG), the CORBA standards body, approved in March 2000. The final specifications for the new Fault-Tolerant CORBA standard correspond closely to Eternal's fault tolerance infrastructure.

Acknowledgements

This research has been supported by the Defense Advanced Research Projects Agency in conjunction with the Office of Naval Research and the Air Force Research Laboratory, Rome, under Contracts N00174-95-K-0083 and F3602-97-1-0248, respectively.

References

[1] K. P. Birman and R. van Rennesse. *Reliable Distributed Computing Using the Isis Toolkit.* IEEE Computer Society Press, 1994.

[2] M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, October 1998.

[3] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[4] Isis Distributed Systems Inc. and Iona Technologies Limited. *Orbix+Isis Programmer's Guide*, 1995.

[5] S. Maffeis. Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, pages 135–146, Monterey, CA, 1995.

[6] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni. An interoperable replication logic for CORBA systems. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 7–16, Antwerp, Belgium, September 2000.

[7] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and implementation of a CORBA fault-tolerant object group service. In *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Helsinki, Finland, June 1999.

[8] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[9] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[10] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 507–516, Austin, TX, May 1999.

[11] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 39–48, Antwerp, Belgium, September 2000.

[12] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.3 edition. OMG Technical Committee Document formal/98-12-01, June 1999.

[13] Object Management Group. Fault tolerant CORBA (final adopted specification). OMG Technical Committee Document ptc/2000-04-04, March 2000.

[14] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.

[15] A. Vaysburd and K. Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.